V for Virtual

Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 162 (2006) 177–181
www.elsevier.com/locate/entcs

V for Virtual

Andrew D. Gordon

Microsoft Research

Abstract

Operating system virtualization has been available on commodity hardware for a few years, and today attracts considerable commercial and research interest. Virtualization allows one or more virtual machines (VMs) to run on a single physical machine, and to interact via virtual devices, such as virtual hard discs or virtual network cards. To model basic virtualization operations, we propose a process calculus, V, with primitives to start and stop VMs, and to read and write data in a hierarchical store. Formalisms such as V may be useful for programming and reasoning about various applications of virtualization, such as VM-based trusted computing or VM-based computational grids.

Keywords: Operating system virtualization; formalization.

Operating system virtualization allows a host operating system, running directly on a physical machine and controlling its devices, to run multiple guest operating systems within virtual machines. The virtualization software, known as a *hypervisor* or a *virtual machine monitor* (VMM), may run under the host operating system as an application (for example, Virtual PC [6] under Windows), or it may be the host operating system itself (for example, Xen [1]).

Following research in the 1960s, IBM launched the first commercial VMM in 1972: VM/370 manages an IBM System 370 mainframe and gives each user at a terminal the impression they have a complete System 370. VMware launched the first commercial VMM for desktop PCs in 1999. Since then several VMMs for the x86 PC architecture have appeared, aimed both at desktops and server farms. Today, OS virtualization is increasingly mobile: a suspended VM together with its *virtual hard disc* (VHD) file are typically several gigabytes, but comfortably fit in say a disc-based personal music player, not to mention a laptop. VMMs on devices such as phones and PDAs cannot be far off.

Virtualization has many applications. Parallelism between VMs enables better utilization of physical assets: applications in different guest operating systems share physical resources. A legacy application on a legacy guest operating system can run on new hardware in a new host operating system. Isolation between VMs enables security mechanisms [5] and enables debugging to proceed in parallel with

doi:10.1016/j.entcs.2006.01.030

production (a significant attraction of VM/370). Creation of fresh VMs and VHDs enables *disposable computing*: creation of a virtual computer to run beta code or code suspected of bearing spyware to be uninstalled reliably just by deleting the VM and VHD. A VM-based honeypot makes a disposable VM for each incoming network probe. Checkpointing and restarting of VM and VHD state enables several applications: load balancing via live migration; analysis of saved state for forensic purposes such as debugging or intrusion-detection; and pre-packaged training demos—last but not least.

1 A Calculus of Virtual Machines and Virtual Discs

This paper introduces V, a formalism for describing typical usages of VMs attached to VHDs. V is based on named, copyable processor *partitions* interacting via a global, hierarchical store. The partitions model both physical and virtual machines, while the global store models the file store of the host operating system, including attached VHDs.

Syntax of Values, Stores, and Processes:

$U, V ::=$	storable value
$\quad x, y, z$	variable
$\quad a, b, c$	name
$\quad U/V$	path construction
$\quad S$	store
$\quad \mathbf{proc}(P)$	process
$S ::= \{a_1{=}V_1, \ldots, a_n{=}V_n\}$	store: a_i pairwise distinct, $V_i \neq \{\}$, V_i closed
$P, Q ::=$	process
$\quad \mathbf{new}\ a; P$	name restriction
$\quad P \mid Q$	composition
$\quad U[P]$	partition named U enclosing process P
$\quad \mathbf{write}(U, V); P$	write value V at path U
$\quad \mathbf{let}\ x = \mathbf{read}(U)\ \mathbf{in}\ P$	read value x from path U
$\quad \mathbf{run}(U)$	run code U
$\quad \mathbf{let}\ x = \mathbf{stop}(U)\ \mathbf{in}\ P$	stop partition named U, save state as x
$C ::= P\ S$	configuration

In the phrase $\mathbf{new}\ a; P$, the name a is bound with scope P. In the phrases $\mathbf{let}\ x = \mathbf{read}(U)\ \mathbf{in}\ P$ and $\mathbf{let}\ x = \mathbf{stop}(U)\ \mathbf{in}\ P$, the variable x is bound with scope P. We say a phrase of syntax is *closed* if and only if it contains no variables.

We assume each V_i in a store $\{a_1{=}V_1, \ldots, a_n{=}V_n\}$ is closed and distinct from $\{\}$. We use the empty store $\{\}$ as a distinguished *null* value. Let a *path* be either null $\{\}$, or p/a where p is a path and a is a name. Hence, a path is a possibly-empty list of names. We often omit the initial $\{\}$. We write $p@p'$ for the path obtained by concatenating paths p and p'. Processes perform non-blocking reads and writes of values at a path in the store. Null cannot occur as an explicit value in a store, but

reading from a non-existent path returns null, and writing null to a path amounts to deletion of the previous contents of the path.

To isolate named partitions, each process interacts with the global store relative to a path. A *configuration* $P\ S$ is a snapshot of a whole computation, consisting of a top-level process P running at path $\{\}$ relative to global store S. In a configuration $P \mid a[Q]\ \{a = S_a, b = S_b\}$, P runs at $\{\}$ and sees the whole store $\{a = S_a, b = S_b\}$, while Q runs at $/a$ and so sees just S_a.

Next, we describe the semantics of a process P running at a path r relative to an implicit global store. The restriction **new** $a; P$ at r creates a fresh name a and behaves as P at r. The composition $P \mid Q$ at r is the parallel composition of processes P and Q running at r. The partition $a[P]$ at path r encloses the process P running at r/a. The process $\mathbf{write}(p, V); Q$ running at r deposits V into the store at $r@p$, then behaves as Q at r. The process **let** $x = \mathbf{read}(p)$ **in** Q running at r retrieves the value V at $r@p$ from the store, then behaves as $Q\{x \leftarrow V\}$ at r. The process $\mathbf{run}(\mathbf{proc}(P))$ running at r behaves the same as P at r. The process **let** $x = \mathbf{stop}(a)$ **in** Q at r blocks until there is a partition $a[P]$ directly in parallel, stops it, then behaves as $Q\{x \leftarrow \mathbf{proc}(a[P])\}$ at r.

Below, we use the shorthand $\mathbf{done} \triangleq \mathbf{run}(\{\})$ for a stuck, terminal process.

2 Using V to Model Operations on Virtual PCs

For the purpose of a simple example, let a *VPC* be the virtualization of a processor coupled with a single bootable disc. This is a common case in desktop uses of virtualization.

Our model mimics one particular VMM [6] and stores the state of a VPC in three files managed by the host operating system. A file MyVPC.vhd holds the VHD, the image of the whole file system available to the guest operating system. A file MyVPC.vsv contains the state of the suspended VM. A file MyVPC.vmc is an XML database containing the configuration of the VPC, including paths to MyVPC.vhd and MyVPC.vsv.

Hence, we model an inactive VPC with a guest file system S and current state P as a store containing three such files. The name vm007 is a unique identifier for the VPC.

{ MyVPC.vhd = S, MyVPC.vsv = **proc**(vm007[P]),
MyVPC.vmc = {id=vm007, disc=/MyVPC.vhd, mem=/MyVPC.vsv} }

To activate a VPC, we copy the VHD S to a temporary file vm007, and run the partition vm007[P], so that P sees S as its store. After the partition and store have run for a while, and evolved to say vm007[P'] and S', the configuration takes the general form:

vm007[P']
{ vm007 = S', MyVPC.vhd = S, MyVPC.vsv = **proc**(vm007[P]),
MyVPC.vmc = {id=vm007, disc=/MyVPC.vhd, mem=/MyVPC.vsv} }

We show some V processes to create, activate, and stop VPCs; for simplicity, we omit synchronization code. Let a *bootable VHD* be a store with a file at /boot.exe containing a process that initializes the guest operating system. Given a bootable VHD at path vhd, the following creates an inactive VPC, by storing its state and configuration at paths vsv and vmc:

newVM vhd vsv vmc $\triangleq$
new vm; **write**(vsv, **proc**(vm[**let** x = **read**(/boot.exe) **in run**(x)]));
write(vmc, {id=vm, disc=vhd, mem=vsv}); **done**

The following activates an inactive VPC at path vmc:

startVM vmc $\triangleq$
let i = **read**(vmc/id) **in**
let vhd = **read**(vmc/disc) **in let** d = **read**(vhd) **in write**(/i,d);
let vsv = **read**(vmc/mem) **in let** m = **read**(vsv) **in run**(m)

We present two ways of stopping an active VM. The first simply deletes the running instance, leaving the original VHD and image files intact, while the second updates the files with the current VHD and machine state. Both write {} to delete the temporary VHD copy.

stopAndDeleteChanges vmc $\triangleq$
let i = **read**(vmc/id) **in**
let m = **stop**(i) **in write**(/i,{}); **done**
stopAndSaveChanges vmc $\triangleq$
let i = **read**(vmc/id) **in**
let m = **stop**(i) **in let** d = **read**(/i) **in write**(/i,{});
let vsv = **read**(vmc/mem) **in write**(vsv,m);
let vhd = **read**(vmc/disc) **in write**(vhd,d); **done**

3 Conclusion and Future Research

We propose V as a simple formalism for modelling OS virtualization. V is more expressive than the examples of this paper may indicate; we can encode iteration, Booleans and conditionals, VM checkpointing, and various synchronization and communication operations. Perhaps V can itself be encoded within some existing process calculus; it certainly has features in common with many, including the higher-order π-calculus [7], the ambient calculus [2], the seal calculus [3], and Xdπ [4]. A formal theory of V, together with an implementation over a VMM, would be a useful first assessment of the calculus.

OS virtualization is an old technology, but its emergence on commodity hardware enables new and complex applications. One example is trusted computing based on attestation of software isolated within a VM, as in Terra [5] or Microsoft NGSCB, for instance. Formalisms like V, extended perhaps with symbolic cryptography, would enable formal security analyses of such applications.

Another example is the idea of a *virtual cluster*, an application built from component VMs running applications like web servers and databases, and interconnected by virtual networks. Virtual clusters consisting of tens, hundreds, or more VMs are envisaged as an efficient way to utilize large data centres. The lifecycle of a virtual cluster is complex and long-lasting; to minimise costly operator intervention, programs controlling virtual clusters should automatically handle events such as VM failure, checkpointing and restarting, automatic contraction and expansion of the size of the virtual cluster, load balancing VMs between physical hardware, and so on. Conventional testing of scripts controlling virtual clusters will likely prove inadequate in finding bugs—many critical error conditions seldom occur. So, we should investigate programming techniques, perhaps prototyped in calculi such as V, for building virtual cluster control software that is amenable to static analysis.

Acknowledgement

Conversations with Paul Barham, Nick Benton, Beppe Castagna, Philippa Gardner, and Ant Rowstron were useful.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, 2003.

[2] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000.

[3] G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *Information and Computation*, 201(1):1–54, 2005.

[4] P. Gardner and S. Maffeis. Modelling dynamic web data. *Theoretical Computer Science*, 342(1):104–131, 2005.

[5] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Symposium on Operating Systems Principles (SOSP'03)*, pages 193–206, 2003.

[6] Microsoft Corporation. Microsoft Virtual PC 2004. Product web page, at http://www.microsoft.com/windows/virtualpc/default.mspx.

[7] D. Sangiorgi. *Expressing mobility in process algebras: first-order and higher-order paradigms*. PhD thesis, University of Edinburgh, 1993.